

Introduction to Python

<https://forge.uclouvain.be/barriat/learning-python>



October 16, 2024

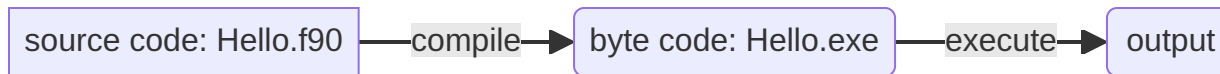
ELIC Training Sessions

Programming basics

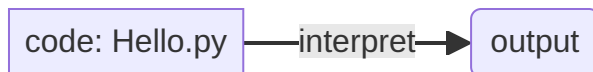
- **code or source code:** The sequence of instructions in a program.
- **syntax:** The set of legal structures and commands that can be used in a particular programming language.
- **output:** The messages printed to the user by a program.
- **console:** The text box onto which output is printed.

Compiling and interpreting

Many languages require you to compile (translate) your program into a form that the machine understands.



Python is instead directly interpreted into machine instructions.



Python: Overview

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Simple syntax, relatively easy to learn
- Useful in many areas (science, web development, GUI programming)
- Big standard library, many additional packages
- Python 3: new minor version (e.g. 3.9) released every October
- Python 2: support ended in 2019, 10% of developers were still using it

Code examples

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print('print me')
print me
>>> # this is a comment
>>> a = 3
>>> b = 4
>>> a * b
12
```

Results of expressions are printed on the screen

Running Python code

- write a program as a file (or collection of files), run that program
| GUI applications, web applications, data processing pipelines
- type code into an interactive console or notebook line by line
| for quick calculations, experimenting, data exploration / analysis)

Using Python

- local installation
- remote Python server
- online Python consoles or online Notebooks (Jupyter)

| <https://www.python.org/shell/>

| <https://jupyterhub.cism.ucl.ac.be>

Expressions

- **expression:** a data value or set of operations to compute a value.

`1 + 4 * 3`

- Arithmetic operators we will use:
 - `+ - * /` addition, subtraction/negation, multiplication, division
 - `%` modulus, a.k.a. remainder
 - `**` exponentiation
- **precedence:** order in which operations are computed.
 - `/ % **` have a higher precedence than `+ -`

`1 + 3 * 4` is `13`

`(1 + 3) * 4` is `16`

Variables

- **Variable:** a named piece of memory that can store a value.
 - Compute an expression's result,
 - store that result into a variable,
 - and use that variable later in the program.
- **Assignment statement:** stores a value into a variable
 - Syntax: `name = value`
 - Examples: `x = 5` , `gpa = 3.14`
 - A variable that has been given a value can be used in expressions.
`x + 4` is `9`

- Names of variables are usually written in lower case, separating words by underscores

```
birth_year = 1970
current_year = 2020
age = current_year - birth_year
```

- Variable names may only consist of letters, digits and underscores
- Overwriting (reassigning) variables:

```
name = "John"
name = "Jane"
a = 3
a = a + 1
```


Basic (primitive) data types

- `int` (integer)
- `float` (floating point number)
- `str` (string): text
- `bool` (boolean): yes / no
- `none`: missing / unknown value

Strings can be enclosed in single or double quotes

```
greeting = "Hello"  
name = 'John'
```

Inserting a variable (f-strings):

```
message1 = f"Hello, {name}!"
```

Joining strings:

```
message2 = "Hello, " + name + "!"
```


Strings - escape sequences

```
text = "He said: \"hi!\""
```

Line break: `\n`

```
a = 'line 1\nline 2'
```

single Backslash: `\\`

```
b = 'C:\\docs'
```


boolean value: yes/no

In Python: `True` or `False`

| note the capitalization

None represents a value that is unknown or missing

```
first_name = "John"  
middle_name = None  
last_name = "Doe"
```


Integer division

- When we divide integers with `/`, the quotient is also an integer.
 - `35 / 5` is `7`
 - `218 / 5` is `43`
 - `156 / 100` is `1`
- The `%` operator computes the remainder from a division of integers.
 - `218 % 5` is `3`
 - `84 % 10` is `0`

Real numbers

- Python can also manipulate real numbers.

`6.022` , `-15.9997` , `42.0` , `2.143e17`

- The operators `+` `-` `*` `/` `%` `**` `()` all work for real numbers.

- The `/` produces an exact answer: `15.0/2.0` is `7.5`

- The same rules of precedence also apply to real numbers: Evaluate `()` before `*` `/` `%` before `+` `-`

- When integers and reals are mixed, the result is a real number.

- Example: `1/2.0` is `0.5`

- The conversion occurs on a per-operator basis.

Types and type conversions

Determining the type of a variable via `type` :

```
a = 4 / 2  
  
type(a)
```

Objects may be converted to other types via `int()` , `float()` , `str()` , `bool()` , ...

```
pi = 3.1415  
pi_int = int(pi)  
message = "Pi is approximately " + str(pi_int)
```


- `int(x)` converts `x` to an integer
- `float(x)` converts `x` to a floating point
- The interpreter shows a lot of digits

```
>>> 1.23232
1.232320000000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```


Functions

A function is a "sub-program" that can perform a specific task

Examples of predefined functions:

- `len()` can determine the length of a string (or of a list, ...)
- `id()` can determine the internal ID of an object
- `type()` can tell us the type of an object
- `print()` can write some output into the terminal
- ...

- A **function** can receive so-called *parameters* and produce a result (a *return value*)
 - `len()` can take a string as a *parameter* and produce an int as the *return value*
 - `print()` can take various objects as *parameters*; it does **not** have an explicit *return value*
- A **method** is a function that belongs to a specific object type (e.g. to *str*)

Examples of **string methods**:

- `first_name.upper()`
- `first_name.count("a")`
- `first_name.replace("a", "@")`

Builtins, standard library

- **Builtins:** functions and objects that are used frequently and are available at all times
- **Standard library:** collections of additional modules and packages that can be imported

Documentation: <https://docs.python.org/3/library/index.html>

Builtins

Amongst others: `print()`, `input()`, `len()`, `open()`, etc

Standard library

The standard library contains additional modules that can be imported.

Example:

```
import math  
print(math.floor(3.6))
```

or

```
from math import floor  
print(floor(3.6))
```


Math functions

Python has useful functions for performing calculations.

Function name	Description
<code>ceil(value)</code>	rounds up
<code>floor(value)</code>	rounds down
<code>log(value)</code>	logarithm, base e
<code>cos(value)</code>	cosine, in radians
<code>sqrt(value)</code>	square root

etc...

Math constants

Constant	Description
<code>e</code>	2.7182818...
<code>pi</code>	3.1415926...

To use many of these above, you can write the following at the top of your Python program:

```
from math import *
```


Text input/output

- `input` reads a number from user input.

You can assign (store) the result of input into a variable.

- `print` produces text output on the console.

Prints the given text message (or expression value) on the console, and moves the cursor down to the next line:

- `print "Message" , print Expression`

Prints several messages and/or expressions on the same line:

- `print Item1, Item2, ..., ItemN`

A comma at the end will not print a newline character: `print 'hello',`

Examples

```
>>> x = 7
>>> x
7
>>> x+7
14
>>> x = 'hello'
>>> x
'hello'
>>> print "Hello, world!"
Hello, world!
>>> age = 45
>>> print "You have", 65 - age, "years until retirement"
You have 20 years until retirement
```


Hands-on exercise

Write a program called **age.py** which will ask the user for their birth year and will respond with the user's age in the year 2024.

Example:

```
What's your name?  
> PY  
What year were you born?  
> 1982  
Hi PY! You are 42
```


Logic

Many logical expressions use relational operators:

Operator	Meaning	Example	Result
<code>==</code>	equals	<code>1 + 1 == 2</code>	True
<code>!=</code>	does not equal	<code>3.2 != 2.5</code>	True
<code><</code>	less than	<code>10 < 5</code>	False
<code>></code>	greater than	<code>10 > 5</code>	True
<code><=</code>	less than or equal to	<code>126 <= 100</code>	False
<code>>=</code>	greater than or equal to	<code>5.0 >= 5.0</code>	False

Combining comparisons

Logical expressions can be combined with logical operators:

Operator	Example	Result
and	<code>9 != 6 and 2 < 3</code>	True
or	<code>2 == 3 or -1 < 5</code>	True
not	<code>not 7 > 0</code>	False

Selection

if

Executes a group of statements only if a certain condition is true. Otherwise, the statements are skipped.

if/else

Executes one block of statements if a certain condition is True, and a second block of statements if it is False.

if/elif/else

Multiple conditions can be chained with elif ("else if")

Examples

```
gpa = 3.4
if gpa > 2.0:
    print "Your application is accepted."
```

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30 :
    y = x + 30
else :
    y=x
print 'y = ',
print math.sin(y)
```


Hands-on exercise

Write a script that asks the user to input a year and tells them whether that year is a leap year.

The rules for leap years are:

- in general, a year is a leap year if it is divisible by 4 (e.g. 1904 was a leap year)
- exception from the above: if the year is also divisible by 100 it *is not* a leap year (e.g. 1900 was *not* a leap year)
- exception from the exception: if the year is also divisible by 400 it *is* a leap year (e.g. 2000 *was* a leap year)

Hint: "x is divisible by y" in Python: `x % y == 0`

Repetition

The `for` loop

Repeats a set of statements over a group of values.

```
for x in range(1, 6):  
    print x, "squared is", x * x
```

```
1 squared is 1  
2 squared is 4  
3 squared is 9  
4 squared is 16  
5 squared is 25
```


The `range` function

```
range(start, stop [, step])
```

```
for x in range(5, 0, -1):  
    print x  
print "Blastoff!"
```

```
5  
4  
3  
2  
1  
Blastoff!
```


Cumulative loops

Some loops incrementally compute a value that is initialized outside the loop. This is sometimes called a *cumulative sum*.

```
sum = 0
for i in range(1, 11):
    sum = sum + (i * i)
print "sum of first 10 squares is", sum
```

```
sum of first 10 squares is 385
```


while loops

Executes a group of statements as long as a condition is True.

good for indefinite loops (repeat an unknown number of times)

```
x=1
while x < 10 :
    print x
    x=x+1
```


Loop Control Statements

- **break:** Jumps out of the closest enclosing loop
- **continue:** Jumps to the top of the closest enclosing loop
- **pass:** Does nothing, empty statement placeholder

```
a = 1
while True:
    a = a * 2
    print(a)
    if (a > 1000):
        break
```


Hands-on exercise

Someone opens a new bank account and deposits 100€ at the start of each year.
At the end of a year, they get 4% interest.

How much do they have after 10 years?

Composite types

dictionaries are mappings that contain "named" entries with associated values.

```
person = {  
    "first_name": "John",  
    "last_name": "Doe",  
    "nationality": "Canada",  
    "birth_year": 1980  
}
```

Retrieving and setting elements:

```
person["first_name"]
```

```
person["first_name"] = "Jane"
```


A **list** represents a sequence of objects

```
primes = [2, 3, 5, 7, 11]
users = ["Alice", "Bob", "Charlie"]

products = [
    {"name": "iPhone 12", "price": 949},
    {"name": "Fairphone", "price": 419},
    {"name": "Pixel 5", "price": 799}
]
```

Determining the length

```
len(users)
```

Overwriting a list element

```
users[0] = "Andrew"
```


Retrieving list elements via their index (starting at 0):

```
users[0]  
users[1]  
users[-1] # last element
```

Appending an element

```
users.append("Dora")
```

Removing the last element:

```
users.pop()
```

Removing by index:

```
users.pop(0)
```


Object references and mutations

- What will be the value of `a` after this code has run?

```
a = [1, 2, 3]
b = a
b.append(4)
```

- An assignment (e.g. `b = a`) assigns a new (additional) **name** to an object. The object in the background **is the same**.

If the original should remain intact it may be copied or a derived version can be newly created based on it:

```
a = [1, 2, 3]
# creating a new copy
b = a.copy()
# modifying b
b.append(4)
```

```
a = [1, 2, 3]
# creating a new object b based on a
b = a + [4]
```


- Some objects can be **mutated** (changed) directly

| e.g. via `.append()` , `.pop()` , ...

Examples: `list` , `dict`

- Many simple objects are **immutable** after they have been created.
However, they can be replaced by other objects.

Examples: `int` , `float` , `str` , `bool` , `tuple`

tuple

- Area of application: similar to dicts

```
point_dict = {"x": 2, "y": 4}
point_tuple = (2, 4)
date_dict = {"year": 1973, "month": 10, "day": 23}
date_tuple = (1973, 10, 23)
```

Each entry in a tuple has a specific meaning

- Behavior: similar to lists

```
date_tuple[0] # 1973
len(date_tuple) # 3
```

Unlike lists, tuples are immutable (no `.append` / `.pop` / ...)

Working with files

A **file** is a sequence of bytes on a storage device

Many file formats are a sequence of text characters

e.g. the formats *.txt*, *.html*, *.csv* or *.py*.

The content of text files can be represented as strings (ASCII).

Other file contents can be represented as byte sequences (binary).

Writing a text file

```
file = open("message.txt", "w", encoding="utf-8")  
file.write("hello world\n")  
file.close()
```

The file is opened for writing (w).
The character encoding will be UTF-8.

Reading a text file

```
file = open("message.txt", encoding="utf-8")  
content = file.read()  
file.close()  
print(content)
```

Standard mode: *reading* (r)

File modes

```
# mode: text, append  
open("file.txt", mode="ta")
```

- `t` : text mode (**default**)
- `b` : binary
- `r` : reading (**default**)
- `w` : (over)writing
- `a` : appending

Open and the with statement

```
with open("todos.txt", encoding="utf-8") as file_obj:  
    content = file_obj.read()
```

The file will be closed automatically when the program leaves the indented block.

character encoding

The default character encoding for text files depends on the operating system:

```
import locale  
locale.getpreferredencoding()
```

ASCII, latin1, UTF-8, etc

Recommendation: Use UTF-8 (best support for special characters)

Parts of programs

- programs
 - code blocks
 - statements
 - expressions

Empty code blocks

empty code block via the `pass` statement:

```
# TODO: warn the user if path doesn't exist  
  
if not os.path.exists(my_path):  
    pass
```


Statements across multiple lines

a statement can span across multiple lines if we use parentheses:

```
a = (2 + 3 + 4 + 5 + 6 +  
     7 + 8 + 9 + 10)
```

Alternative: *escaping* newlines with `\`

```
a = 2 + 3 + 4 + 5 + 6 + \  
     7 + 8 + 9 + 10
```


Expressions

expression = something that produces a value (the value might be `None`)

expression = anything that can be on the right-hand side of an assignment (`=`)

examples of expressions:

- `(7 - 3) * 0.5`
- `(7 - 3)`
- `7`
- `round(3.5)`
- `x == 1`

Function parameters

Positional parameters and keyword parameters

Calling `open` :

- with positional parameters:

```
f = open("myfile.txt", "w", -1, "utf-8")
```

- with keyword parameters:

```
f = open("myfile.txt", encoding="utf-8", mode="w")
```


Optional parameters and default parameters

Some parameters of functions can be optional (they have a default value)

Example: For `open` only the first parameter is required, the others are optional

The values of default parameters can be looked up in the documentation

Defining functions

```
def average(a, b):  
    m = (a + b) / 2  
    return m
```

Optional parameters and default parameters

This is how we define default values for parameters:

```
def shout(phrase, end="!"):   
    print(phrase.upper() + end)  
  
shout("hello") # HELLO!  
shout("hi", ".") # HI.
```


Scope

A function definition creates a new **scope**, an area where variables are valid

In the following example there are two distinct variables named `m` :

```
m = "Hello, world"

def average(a, b):
    m = (a + b) / 2
    return m
x = average(1, 2)

print(m) # prints "Hello, world"
```


Scope

Inside a function, outer variables may be read but not overwritten

In other programming languages constructs like `if` or `for` usually also open a new scope - this is not the case in Python

Modules and packages

Module : collection of Python objects that can be imported

Package : collection of modules

packages are actually a special type of modules

- `urllib` = package
- `urllib.request` = module
- `urllib.request.urlopen` = function
- `sys` = module
- `sys.path` = object

Examples:

```
import module1
from package2 import module2a, module2b
from module3 import object3a, object3b
from package4.module4 import object4a, object4b
```

```
import os
from math import sqrt, pi
```

Short names:

```
import numpy as np
import matplotlib.pyplot as plt
```

Importing everything from a module (usually not recommended):

```
from math import *
```


When importing *some* packages, submodules will be imported automatically.

Examples:

```
import os
import numpy as np

os.path.join(...)
np.random.randint(10)
```

Counterexample - this will fail:

```
import urllib

urllib.request.urlopen(...)
```


Conventions for imports

- all imports in a Python file *should* be at the start of the file
- imports *should* be split into three groups:
 - imports from the standard library
 - imports from other libraries
 - imports within the project

Local modules

we can import local Python files as modules

example: local file *messages.py*

```
import messages  
  
print(messages.message1)
```

we can create so-called *packages* as folders

example: folder *phrases/*, files *phrases/messages.py* and *phrases/greetings.py*

```
from phrases import greetings  
  
print(greetings.greeting1)
```


Resolving imports

Search order of imports:

- directory of the Python script that was originally executed
- standard library
- external libraries

Avoid name clashes with existing modules / packages!

NumPy

Library for efficient data processing

Data are stored in multidimensional arrays of numeric values which are implemented in an efficient way:

- smaller memory use than e.g. lists of numbers in Python
- much faster execution of operations like element-wise addition of arrays

Data can represent images, sound, measurements and much more

Common import convention:

```
import numpy as np
```


Pandas

Pandas is a data analysis library; it is based on *NumPy*

```
import pandas as pd
```

Series and DataFrame

- **Series:** Collection of values for some keys (table column)
- **DataFrame:** Collection of associated series (table)

Plotting

Basic (low-level) library for plotting: *matplotlib*

Higher-level interfaces:

- *pyplot* (contained in matplotlib, similar to matlab's plotting interface)
- *pandas* plotting functions (based on pyplot)

Simple plot with pyplot

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([0, 1, 2, 3])

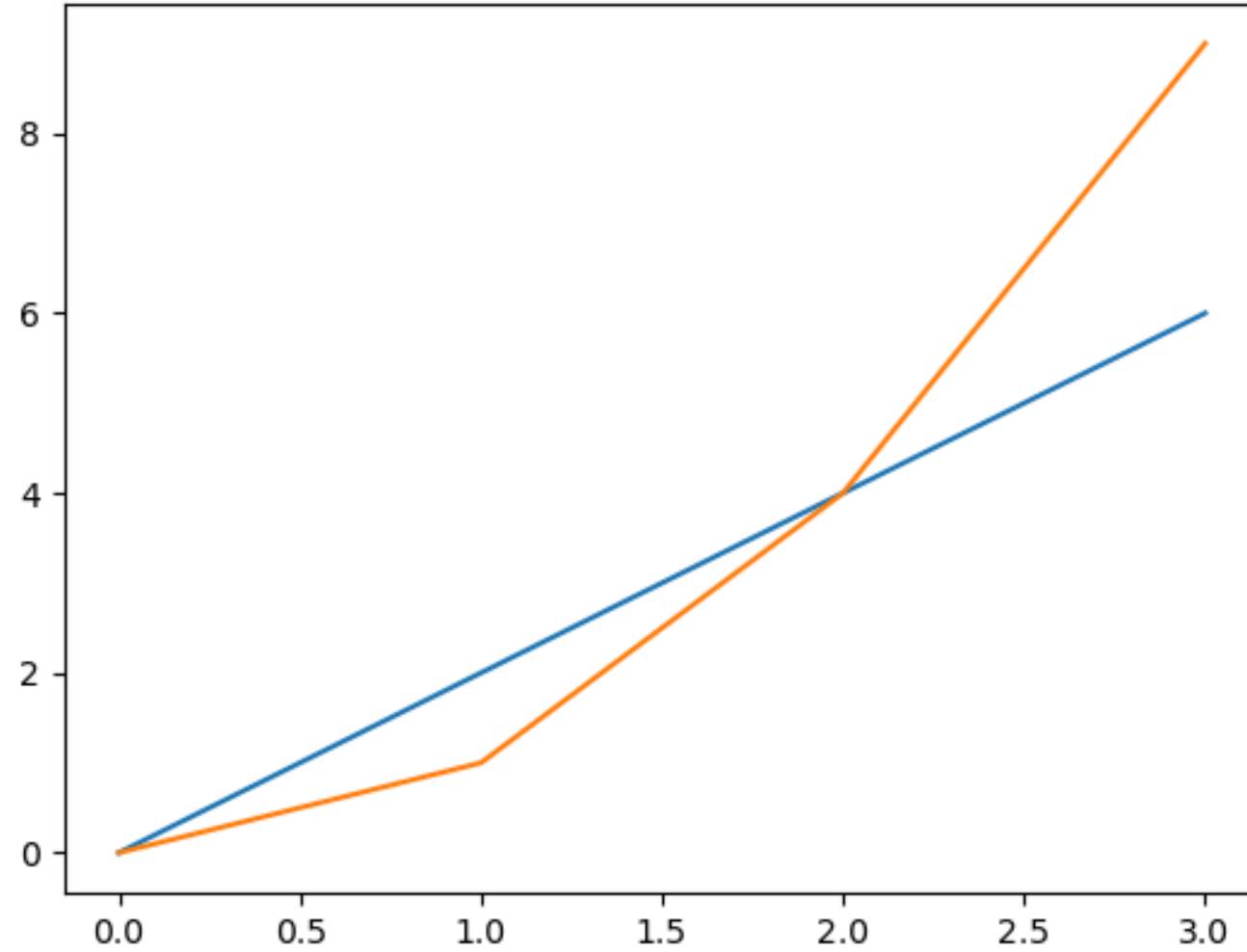
y1 = x*2
y2 = x**2

plt.plot(x, y1)
plt.plot(x, y2)
```

In Jupyter plots are shown automatically

In a regular terminal / program:

```
plt.show()
```

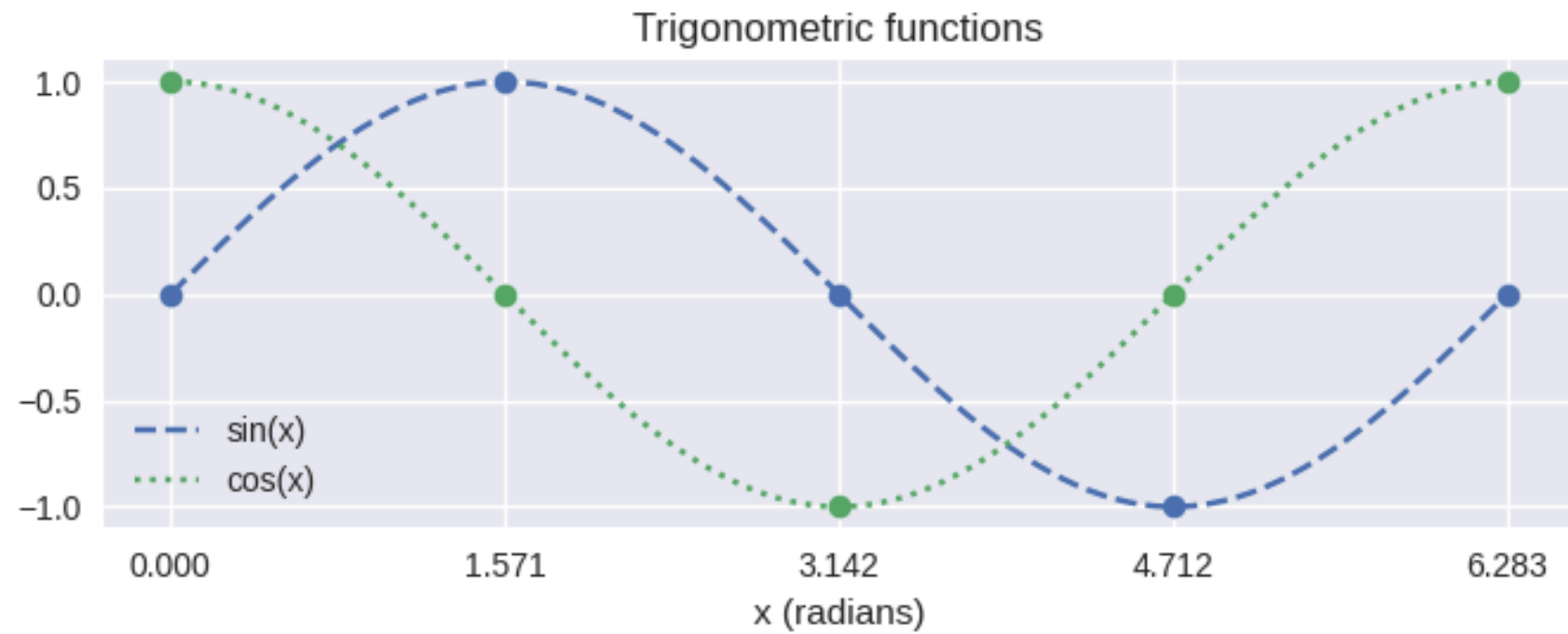



Pyplot: Configuration and Styling

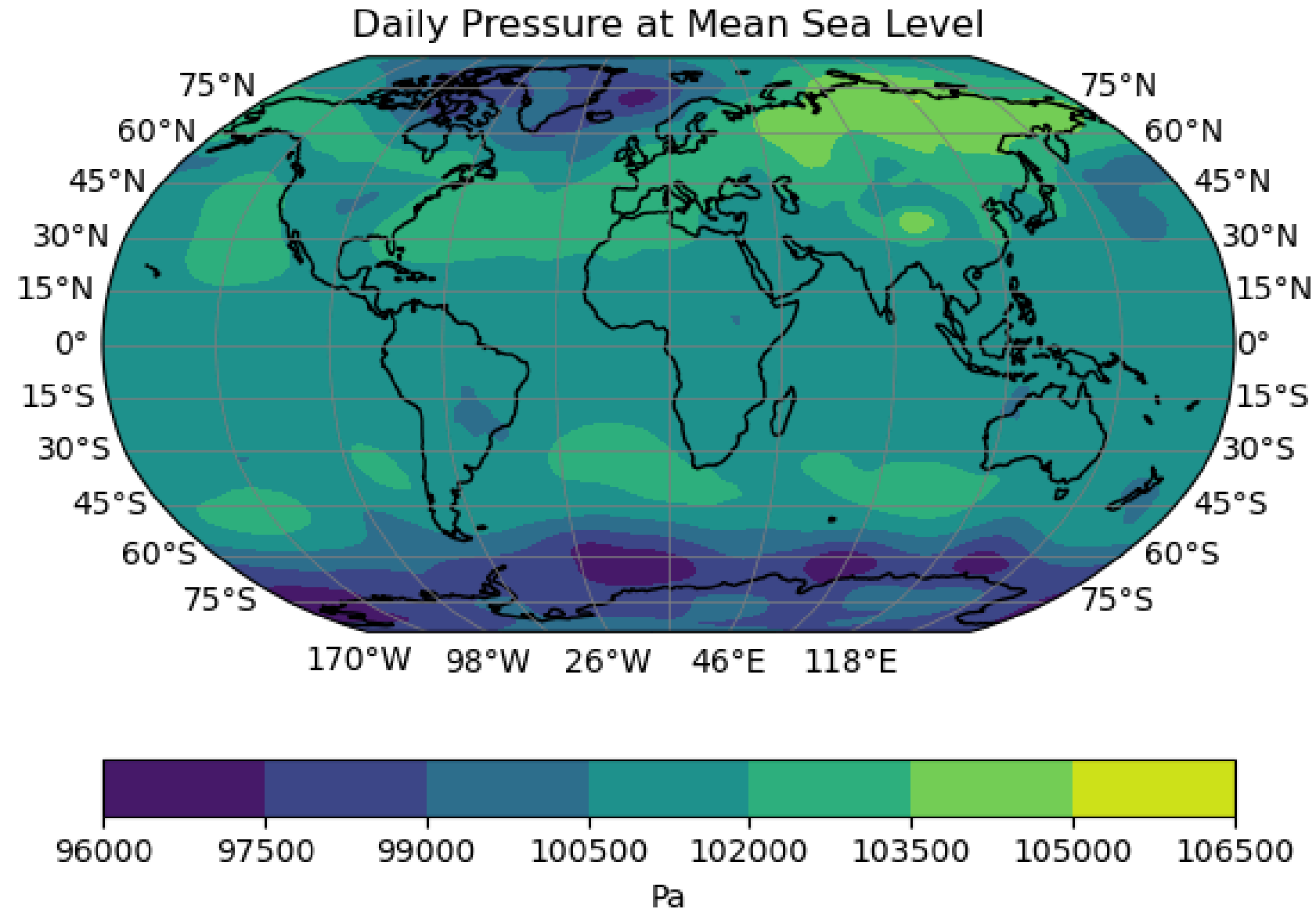
We'll create a plot that shows the sine and cosine functions in the interval from 0 to 2π

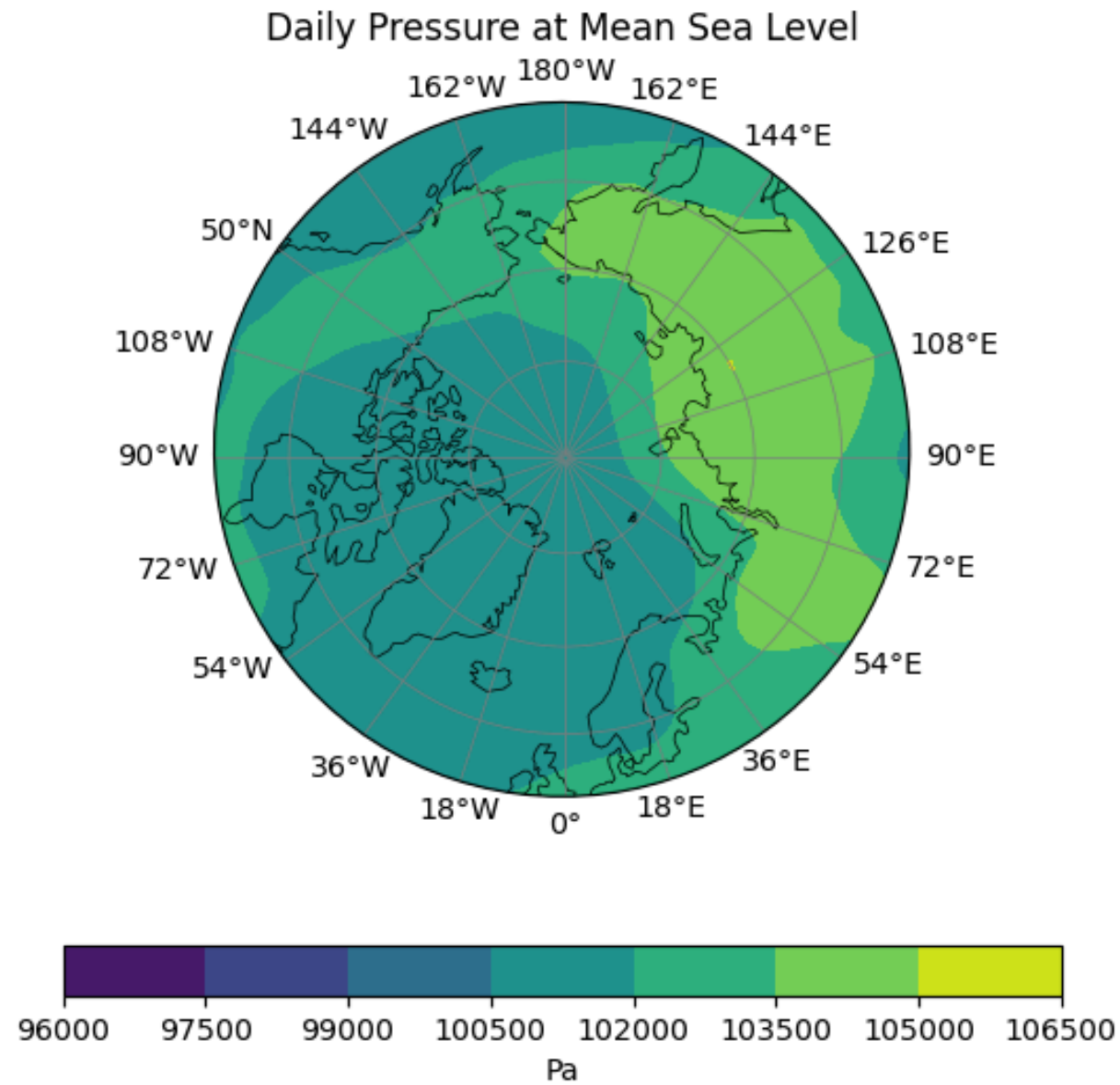
```
x = np.linspace(0, 2*np.pi, 100)

sin = np.sin(x)
cos = np.cos(x)
```

In the following examples we will show how to use **Cartopy** with **netCDF** ClimateData.





Working with various file formats

Possibilities:

- text files
- JSON
- CSV
- XML
- Python object files (via pickle and shelve)
- binary files

JSON

JSON: popular and standardized data file format

can represent the fundamental Python datatypes (none, bool, int, float, list, dict)

Saving JSON:

```
import json
data = ["one", "two", "three"]
jsonstring = json.dumps(data)
with open("numbers.json", mode="w", encoding="utf-8") as jsonfile:
    jsonfile.write(jsonstring)
```

Reading JSON:

```
import json
with open("numbers.json", encoding="utf-8") as jsonfile:
    jsonstring = jsonfile.read()
data = json.loads(jsonstring)
```


CSV

CSV is a file format which can hold tabular data; entries are separated by commas

Example:

```
ISO,Country,Capital,Languages  
AD,Andorra,Andorra la Vella,"ES,FR"  
AE,United Arab Emirates,Abu Dhabi,"AE,fa,en,hi,ur"  
AF,Afghanistan,Kabul,"AF,tk"
```

Python libraries:

- csv (part of the standard library)
- *pandas*

Writing CSV via pandas:

```
import pandas as pd
data = pd.DataFrame(
    [
        ["CN", 9.6, 1386],
        ["RU", 17.0, 144],
        ["US", 9.8, 327],
    ],
    columns=["code", "area", "population"],
)

data.to_csv("countries.csv")
```

Reading CSV via pandas:

```
import pandas as pd
data = pd.read_csv("countries.csv")
print(data)
print(data.values.tolist())
```


Reading and writing CSV

```
import csv

data = [
    ['code', 'area', 'population'],
    ['CN', 9.6, 1386],
    ['RU', 17, 144],
    ['US', 9.8, 327]
]

with open('countr.csv', 'w', encoding='utf-8', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(data)

with open('countr.csv', encoding='utf-8', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```


Thank you for your attention